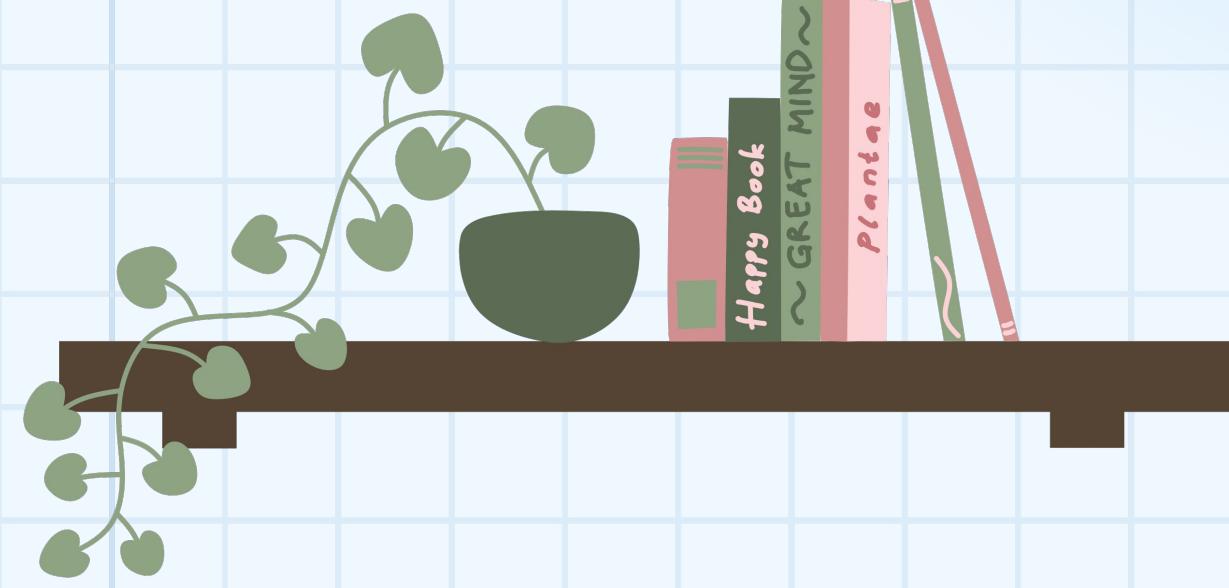
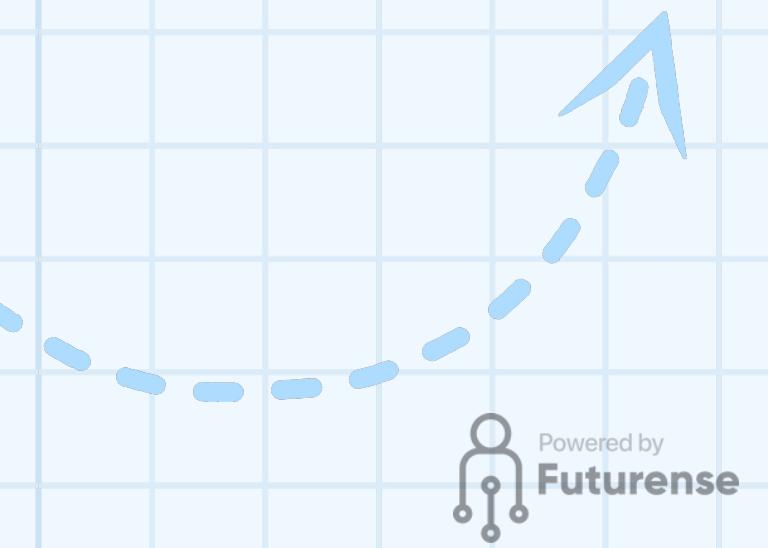
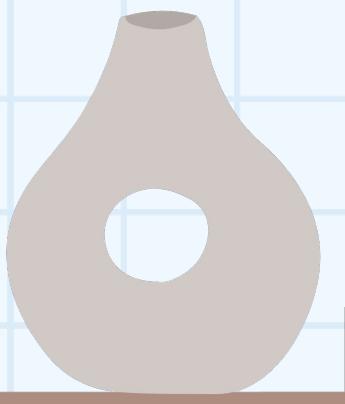
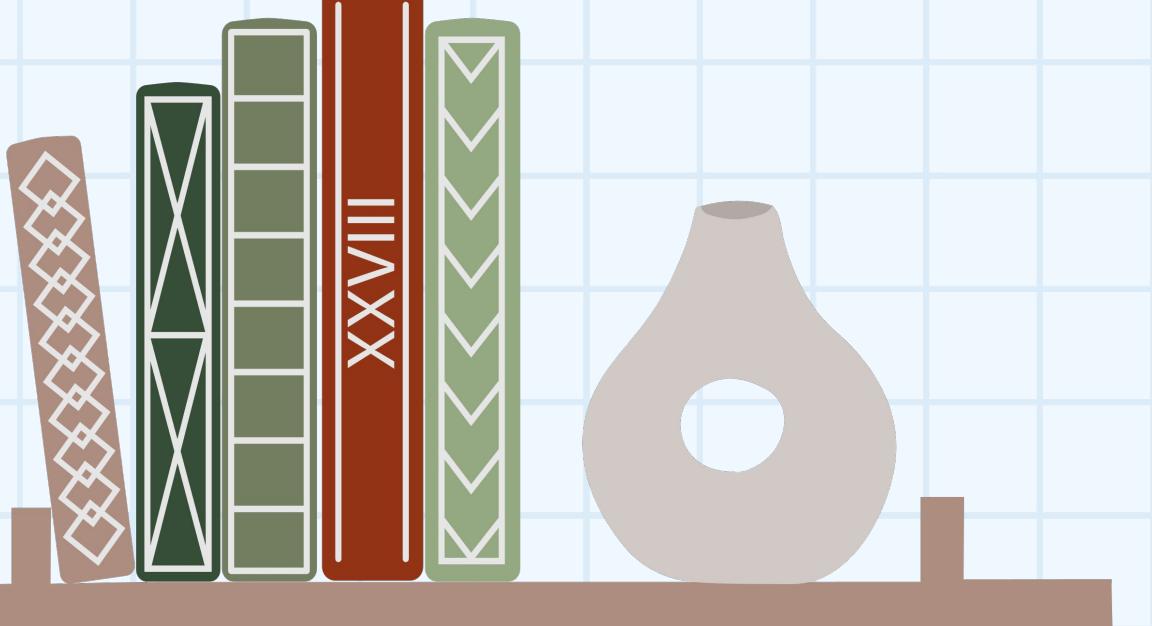




BS./BSC.

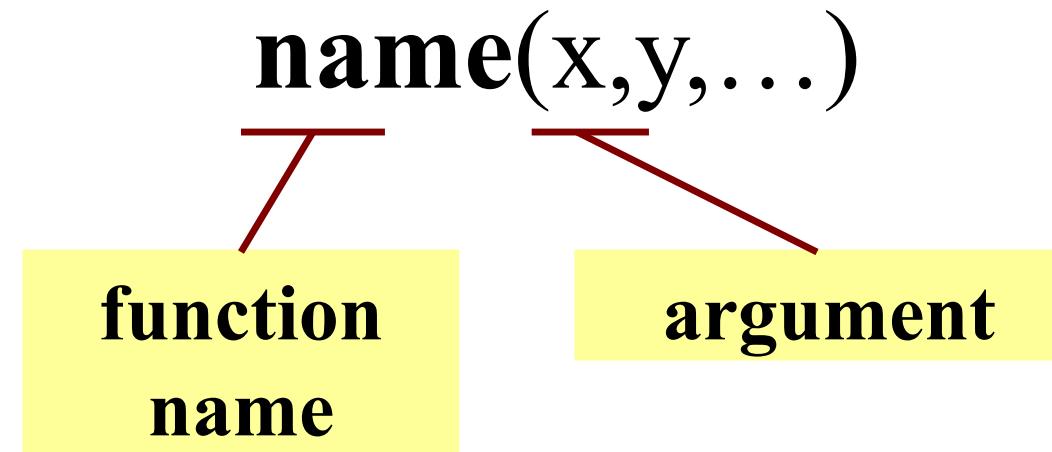
Applied AI and Data Science

Algorithmic Thinking & its Applications



Function Calls

- Python supports expressions with math-like functions
 - A function in an expression is a *function call*
- **Function calls** have the form



- **Arguments** are
 - **Expressions**, not values
 - Separated by commas

Built-In Functions

- Python has several math functions
 - `round(2.34)`
 - `max(a+3,24)`
- You have seen many functions already
 - Type casting functions: `int()`, `float()`, `bool()`
- Documentation of all of these are online
 - <https://docs.python.org/3/library/functions.html>
 - Most of these are too advanced for us right now

Arguments can be
any **expression**

Functions as Commands/Statements

- Most functions are expressions.
 - You can use them in assignment statements
 - **Example:** `x = round(2.34)`
- But some functions are **commands**.
 - They instruct Python to do something
 - Help function:
`help()`
 - Quit function:
`quit()`
- How know which one? Read documentation.



These take no arguments

Built-in Functions vs Modules

- The number of built-in functions is small
 - <http://docs.python.org/3/library/functions.html>
- Missing a lot of functions you would expect
 - **Example:** cos(), sqrt()
- **Module:** file that contains Python code
 - A way for Python to provide optional functions
 - To access a module, the import command
 - Access the functions using module as a *prefix*

Example: Module math

```
>>> import math  
>>> math.cos(0) 1.0
```

```
>>> cos(0)  
Traceback (most recent call last): File "<stdin>",  
line 1, in <module>
```

```
NameError: name 'cos' is not defined  
>>> math.pi 3.141592653589793  
  
>>> math.cos(math.pi)  
-1.0
```

Example: Module math

```
>>> import  
math
```

To access math
functions

```
>>>  
  
math.cos(0)  
  
1.0
```

Functions
require
math
prefix!

```
>>> cos(0)
```

Traceback (most recent call last): File "<stdin>", line 1, in <module>

NameError: name 'cos' is not defined

```
>>> math.pi 3.141592653589793
```

```
>>> math.cos(math.pi)
```

```
-1.0
```

Example: Module math

```
>>> import math  
>>> math.cos(0)      To access math  
                      functions  
  
1.0  
  
>>> cos(0)          Functions  
                      require math  
                      prefix!
```

Traceback (most recent call last): File "<stdin>", line 1, in
<module> NameError: name 'cos' is not defined

```
>>> math.pi           Module has  
                      variables too!  
3.141592653589793  
  
>>> math.cos(math.pi)  
-1.0
```

Example: Module math

```
>>> import math  
>>> math.cos(0)
```

To access math functions

1.0

Functions require math prefix!

```
>>> cos(0)
```

Traceback (most recent call last): File "", line 1, in <module>

NameError: name 'cos' is not defined

```
>>> math.pi
```

3.1415926535897

Module has variables too!

93

```
>>>
```

```
math.cos(math.pi)
```

-1.0

Other Modules

- **os**
 - Information about your OS
 - Cross-platform features
- **random**
 - Generate random numbers
 - Can pick any distribution

Using the from Keyword

```
>>> import  
math
```

Must prefix with
module name

```
>>> math.pi  
3.141592653589793
```

```
>>> from math import pi
```

```
>>>  
pi
```

No prefix needed
for variable pi

```
3.141592653589793
```

```
>>> from math import *
```

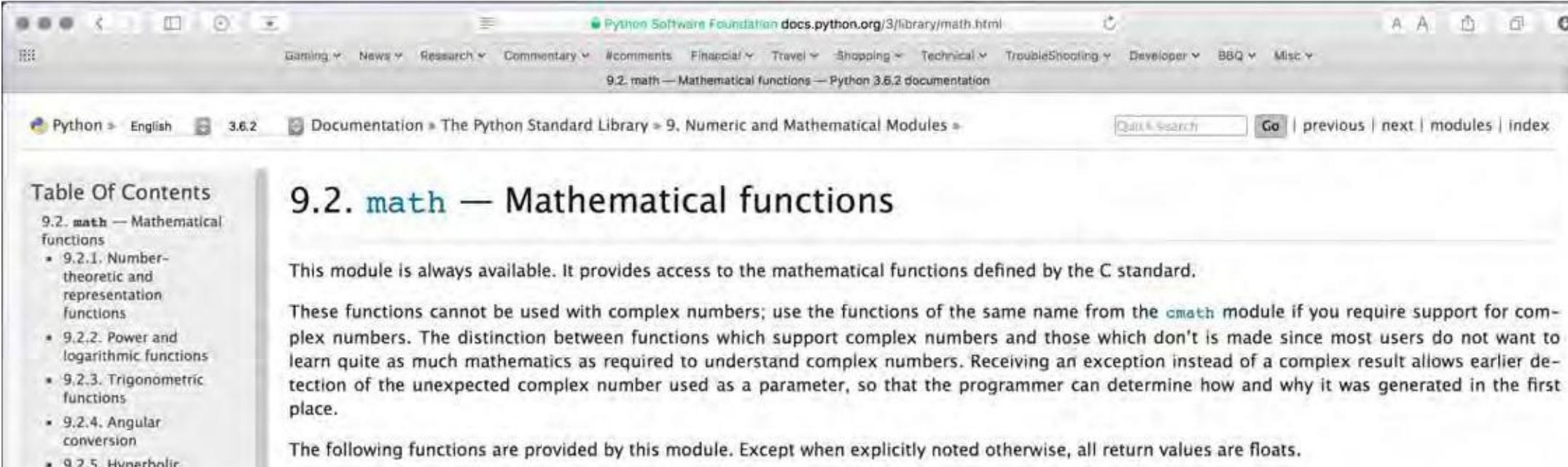
```
>>> cos(pi)
```

```
-1.0
```

No prefix needed for
anything in math

- Be careful using from!
- Using import is *safer*
 - Modules might conflict (functions w/ same name)
 - What if import both?

Reading the Python Documentation



The screenshot shows a web browser displaying the Python 3.6.2 documentation for the `math` module. The title of the page is "9.2. math — Mathematical functions". The left sidebar contains a "Table Of Contents" with sections for Number-theoretic and representation functions, Power and logarithmic functions, Trigonometric functions, Angular conversion, and Hyperbolic functions. The main content area describes the `math` module as always available and providing access to mathematical functions defined by the C standard. It notes that these functions cannot be used with complex numbers and refers to the `cmath` module for support. A note states that the distinction between functions supporting complex numbers and those that don't is made since most users do not want to learn much mathematics. The following functions are listed: `ceil(x)`, `fabs(x)`, `factorial(x)`, `floor(x)`, and `fmod(x, y)`. The `ceil(x)` function is highlighted with a red box.

math. `ceil(x)`

Return the ceiling of x , the smallest integer greater than or equal to x .

Next topic: 9.3. `cmath` — Mathematical functions for complex numbers

This Page: Report a Bug | Show Source

`math. ceil(x)`
Return the ceiling of x , the smallest integer greater than or equal to x .

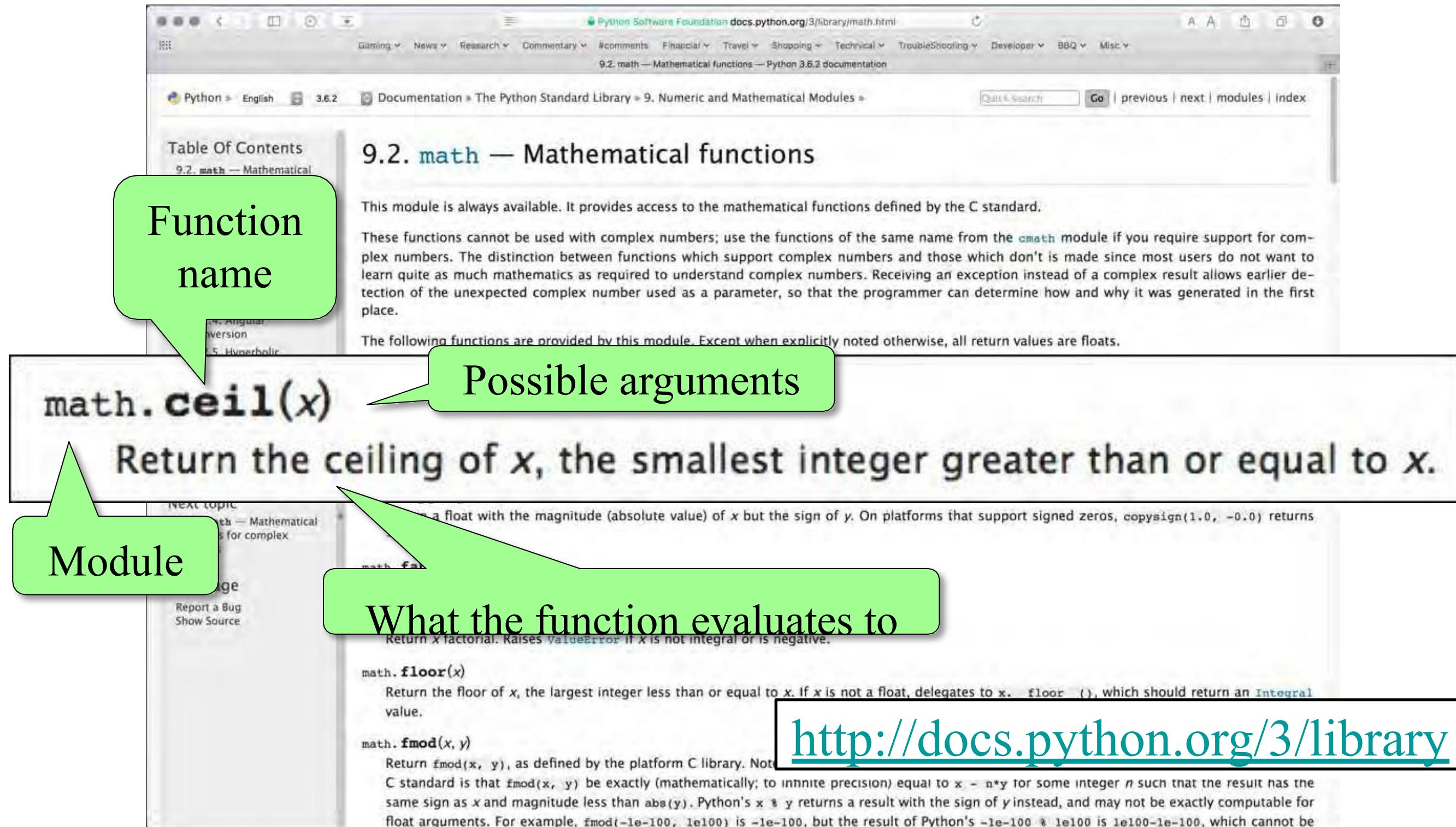
`math. fabs(x)`
Return the absolute value of x .

`math. factorial(x)`
Return x factorial. Raises `ValueError` if x is not integral or is negative.

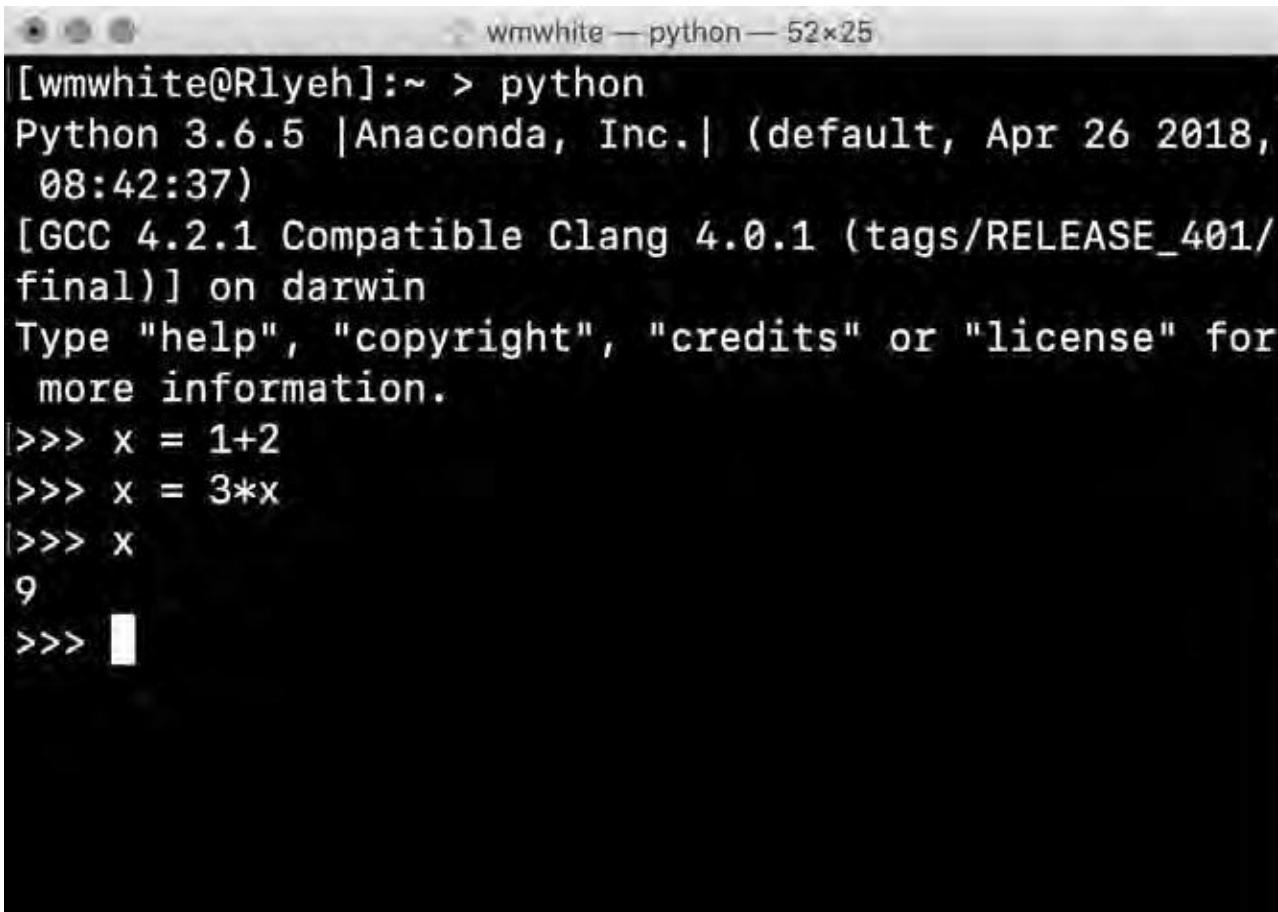
`math. floor(x)`
Return the floor of x , the largest integer less than or equal to x . If x is not a float, delegates to `x.__floor__()`, which should return an `Integral` value.

`math. fmod(x, y)`
Return `fmod(x, y)`, as defined by the platform C library. Note that `fmod(x, y)` be exactly (mathematically; to infinite precision) equal to $x - n*y$ for some integer n such that the result has the same sign as x and magnitude less than `abs(y)`. Python's `x % y` returns a result with the sign of y instead, and may not be exactly computable for float arguments. For example, `fmod(-1e-100, 1e100)` is `-1e-100`, but the result of Python's `-1e-100 % 1e100` is `1e100-1e-100`, which cannot be

Reading the Python Documentation



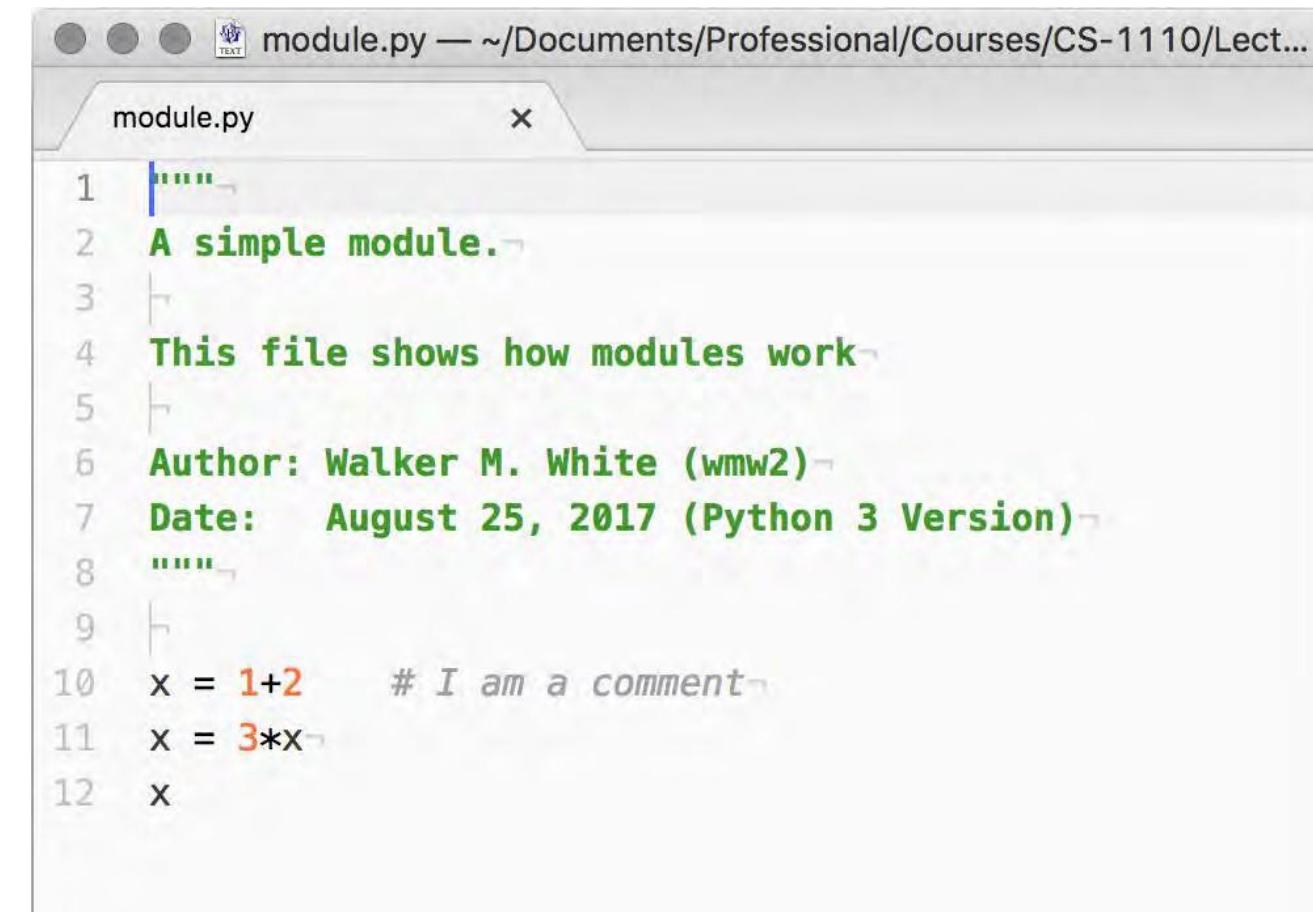
Interactive Shell vs. Modules



```
wmwhite@Riyeh:~ > python
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018,
08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/
final)] on darwin
Type "help", "copyright", "credits" or "license" for
more information.

>>> x = 1+2
>>> x = 3*x
>>> x
9
>>>
```

- Launch in command line
- Type each line separately
- Python executes as you type



```
module.py — ~/Documents/Professional/Courses/CS-1110/Lect...
module.py x

1  """
2  A simple module.
3
4  This file shows how modules work.
5
6  Author: Walker M. White (wmw2)
7  Date: August 25, 2017 (Python 3 Version)
8  """
9
10 x = 1+2    # I am a comment
11 x = 3*x
12 x
```

- Write in a **code editor**
 - We use Pulsar
 - But anything will work
- Load module with import

module.py 1:1 LF UTF-8 MagicPython 0 files 

Using a Module

Module Contents

```
""" A simple module.
```

This file shows how modules
work """

```
# This is a  
comment x = 1+2  
  
x =  
3*x x
```

Using a Module

Module Contents

```
""" A simple module.
```

This file shows how modules
work """

```
# This is a
```

```
comment x = 1+2
```

```
x =
```

```
3*x x
```

Single line comment
(not executed)

Using a Module

Module Contents

```
""" A simple  
module.
```

```
This file shows how modules  
work """
```

```
# This is a  
comment x = 1+2
```

```
x =
```

```
3*x x
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

Single line comment
(not executed)

Using a Module

Module Contents

```
""" A simple  
module.
```

```
This file shows how modules  
work """
```

```
# This is a  
comment
```

```
x = 1+2
```

```
x = 3*x x
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

Single line comment
(not executed)

Commands
Executed on import

Using a Module

Module Contents

```
""" A simple  
module.
```

```
This file shows how modules  
work """
```

```
# This is a  
comment
```

```
x =
```

```
1+2
```

```
x =
```

```
3*x x
```

Docstring (note the Triple Quotes)
Acts as a multiple-line comment
Useful for *code documentation*

Single line comment
(not executed)

Commands

Executed on import

Not a command.
import ignores this

Using a Module

Module Contents Python Shell

""" A simple
module.

This file shows how modules
work """

```
# This is a  
comment x = 1+2  
  
x =  
3*x x
```

```
>>> import  
module
```

```
>>> x
```

Using a Module

Module Contents Python Shell

""" A simple
module.

This file shows how modules
work """

```
# This is a  
comment x = 1+2  
  
x =  
3*x x
```

```
>>> import module  
>>> x  
Traceback (most recent call  
last): File "<stdin>", line 1, in  
<module> NameError: name  
'x' is not defined
```

Using a Module

Module Contents Python Shell

""" A simple
module.

This file shows how modules
work """

This is a comment

x =
1+2

x =

3*x x

**“Module data” must be
prefixed by module name**

```
>>> import module
>>> x
Traceback (most recent call
last): File "<stdin>", line 1, in
<module> NameError: name
'x' is not defined
>>>
module.x 9
```

Using a Module

Module Contents Python Shell

""" A simple module.

This file shows how modules work """

```
# This is a comment
```

```
x =  
1+2
```

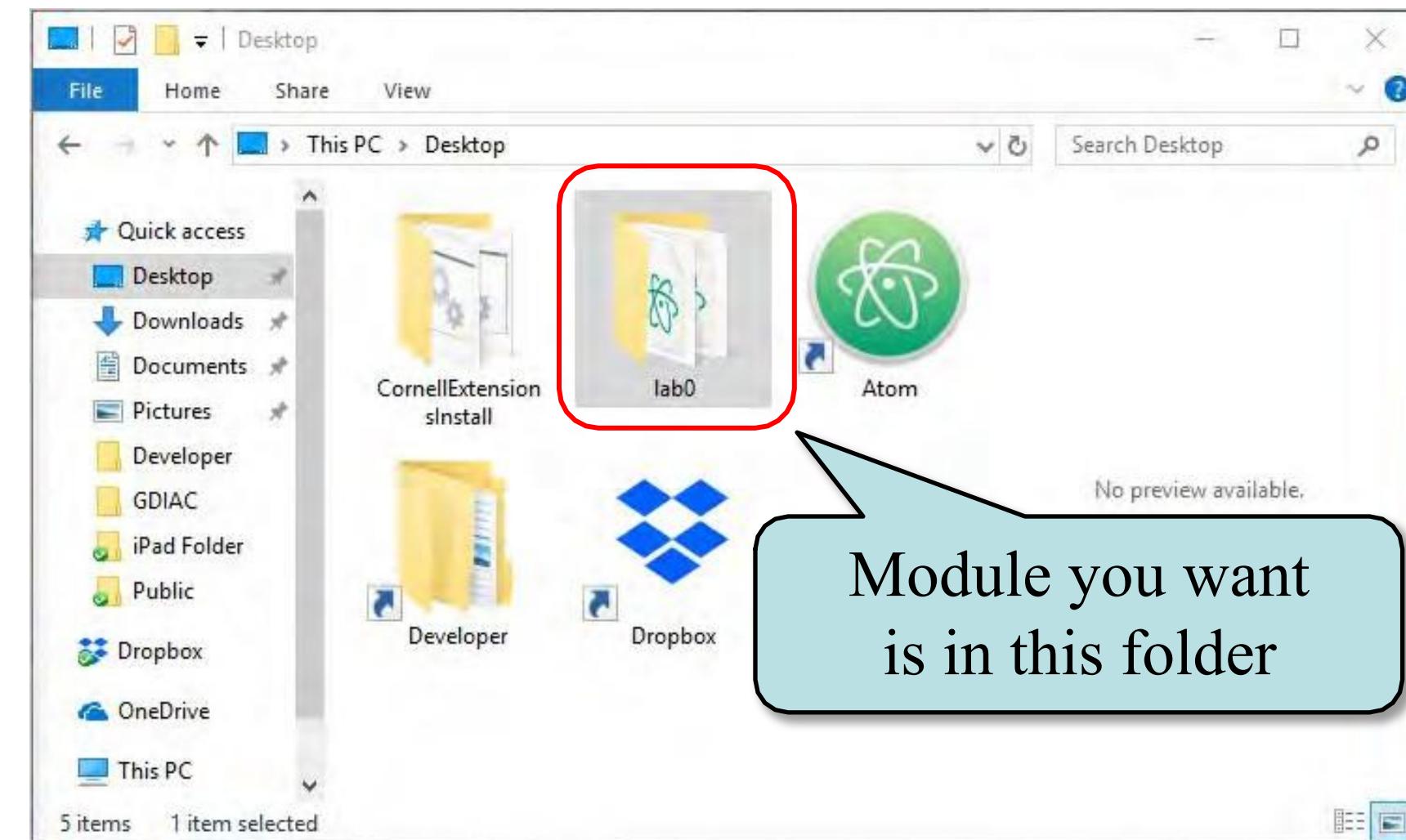
```
x =  
3*x x
```

“Module data” must be prefixed by module name

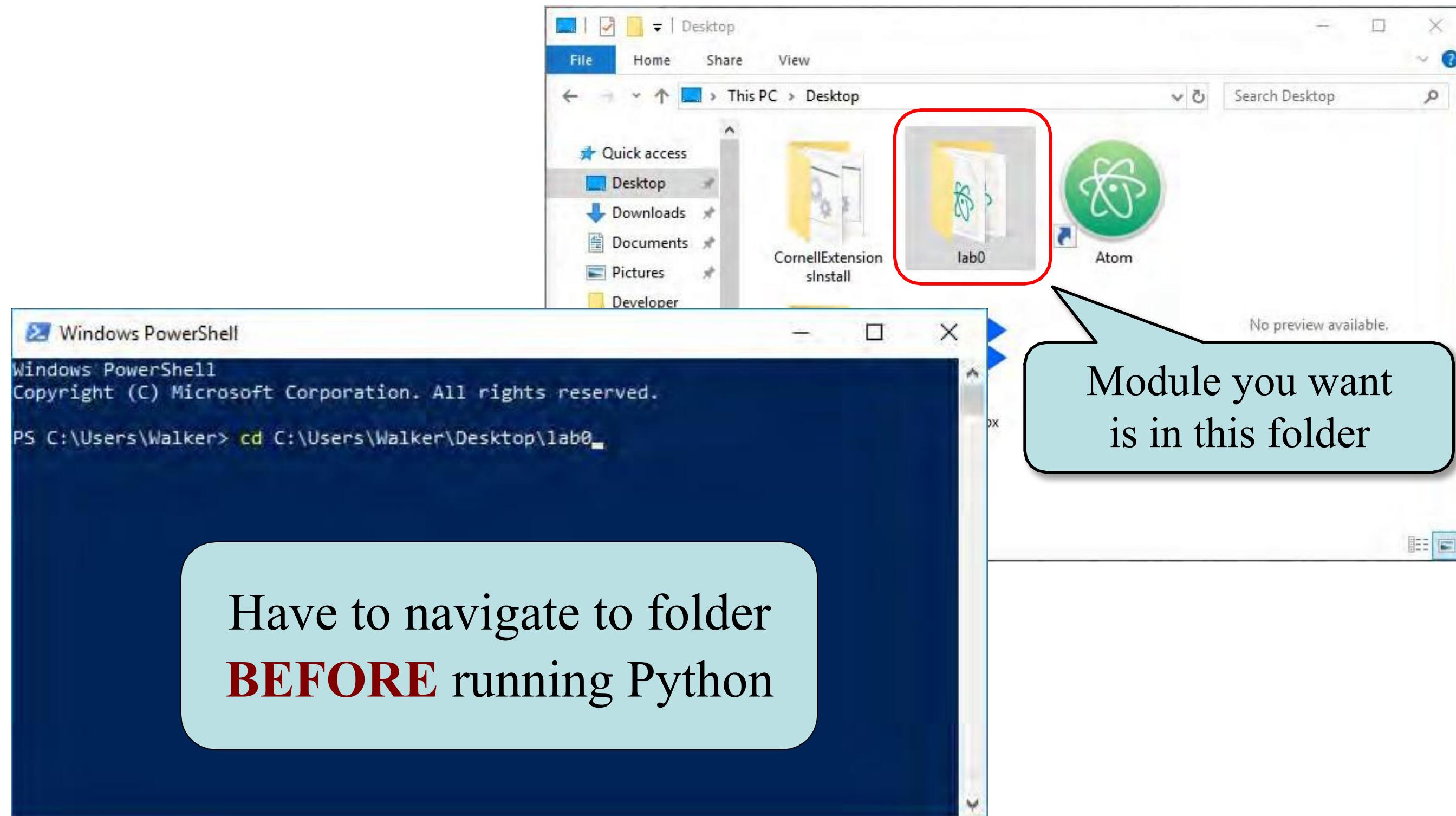
Prints **docstring** and module contents

```
>>> import module  
>>> x  
Traceback (most recent call  
last): File "<stdin>", line 1, in  
<module> NameError: name  
'x' is not defined  
>>>  
module.x 9  
>>> help(module)
```

Modules Must be in Working Directory!



Modules Must be in Working Directory!



Modules vs. Scripts

Module Script

- Provides functions, variables
 - **Example:** temp.py

- import it into Python shell

```
>>> import temp
```

```
>>>
```

```
temp.to_fahrenheit(100)
```

```
212.0
```

```
>>>
```

- Behaves like an application
 - **Example:** helloApp.py

- Run it from command line:

```
python helloApp.py
```



Modules vs. Scripts

Module Script

- Provides functions, variables
 - **Example:** temp.py
- import it into Python shell

```
>>> import temp  
>>>  
temp.to_fahrenheit(100)  
212.0
```

```
>>>
```

Files look the same. Difference is how you use them.

- Behaves like an application
 - **Example:** helloApp.py
- Run it from command line:
python helloApp.py



Scripts and Print Statements

module.py script.py

""" A simple module. """ A simple script.

This file shows how modules
work """

```
# This is a  
comment x = 1+2  
  
x =  
3*x x
```

This file shows why we use
print """

```
# This is a  
comment x = 1+2  
  
x =  
3*x  
print(x  
)
```

Scripts and Print Statements

module.py script.py

""" A simple module. """ A simple script.

This file shows how modules work """

```
# This is a  
comment x = 1+2
```

```
x = 3*x
```

```
x
```



This file shows why we use print """

```
# This is a  
comment x = 1+2
```

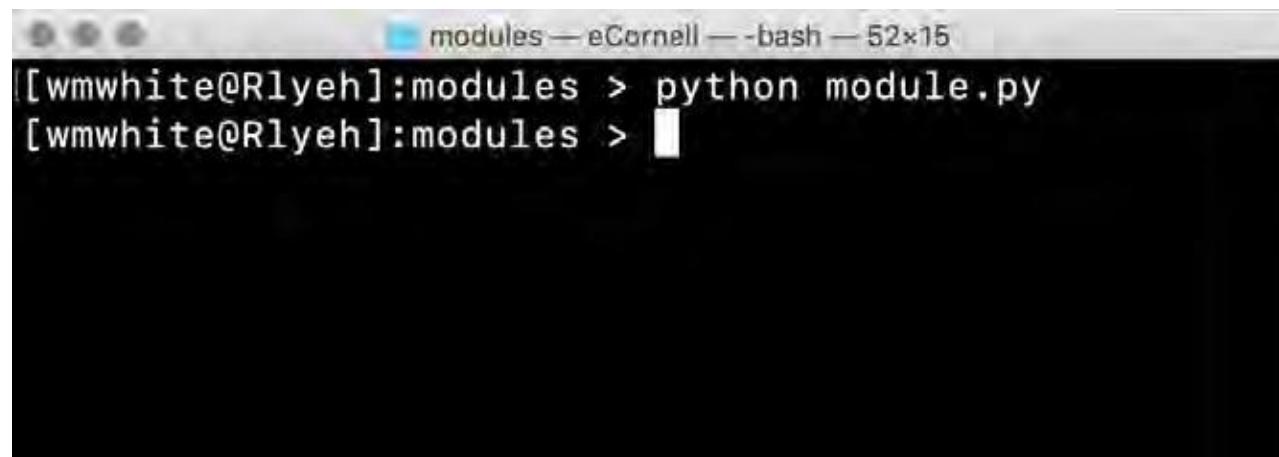
```
x =
```

```
3*x
```

```
print(x  
)
```

Scripts and Print Statements

module.py script.py

A screenshot of a terminal window titled "modules — eCornell — bash — 52x15". The command entered is "[wmwhite@Rlyeh]:modules > python module.py". The output shows the command being run and then the prompt "[wmwhite@Rlyeh]:modules >".

```
[wmwhite@Rlyeh]:modules > python module.py
[wmwhite@Rlyeh]:modules >
```

- Looks like nothing happens
- Python did the following:
 - Executed the **assignments**
 - Skipped the last line
(‘x’ is not a statement)

A screenshot of a terminal window titled "modules — eCornell — bash — 52x15". The command entered is "[wmwhite@Rlyeh]:modules > python script.py". The output shows the command being run, followed by the number "9", and then the prompt "[wmwhite@Rlyeh]:modules >".

```
[wmwhite@Rlyeh]:modules > python script.py
9
[wmwhite@Rlyeh]:modules >
```

- We see something this time!
- Python did the following:
 - Executed the **assignments**
 - Executed the last line
(Prints the contents of x)

Scripts and Print Statements

module.py script.py



The image shows two terminal windows side-by-side. Both windows have a title bar 'modules — eCornell — bash — 52x15' and a command line starting with '[wmwhite@Rlyeh]:modules >'. The left window contains the command 'python module.py' and ends with a blank line. The right window contains the command 'python script.py' followed by the output '9' and a blank line.

When you run a script, only statements are executed. This time!

- Looks like this:
- Python executes the following:
 - Executed the assignments
 - Skipped the last line ('x' is not a statement)
- Executed the assignments
- Executed the last line (Prints the contents of x)

User Input

```
>>> input('Type something')
```

```
Type somethingabc
```

```
'abc'
```

No space after the prompt.

```
>>> input('Type something: ')
```

```
Type something: abc
```

Proper space after prompt.

```
'abc'
```

```
>>> x = input('Type something: ')
```

```
Type something: abc
```

```
>>> x
```

```
'abc'
```

Assign result to variable.

Making a Script Interactive

```
"""
```

A script showing off input.

This file shows how to make a script interactive.

```
"""
```

```
x = input("Give me a  
something: ") print("You  
said: "+x)
```

```
[wmw2] folder> python script.py
```

Give me something: Hello

You said: Hello

```
[wmw2] folder> python script.py
```

Give me something: Goodbye

You said: Goodbye

```
[wmw2] folder>
```

Not using the
interactive shell

Numeric Input

- `input` returns a string
 - Even if looks like int
 - It cannot know better
- You must convert values
 - `int()`, `float()`, `bool()`, etc.
 - Error if cannot convert
- One way to program
 - But it is a *bad* way
 - Cannot be automated

```
>>> x = input('Number: ')
```

Number: 3

```
>>> x  
'3'
```

Value is a string.

```
>>> x + 1
```

```
TypeError: must be str, not int
```

```
>>> x = int(x)
```

```
>>> x+1
```

4

Must convert to
int.

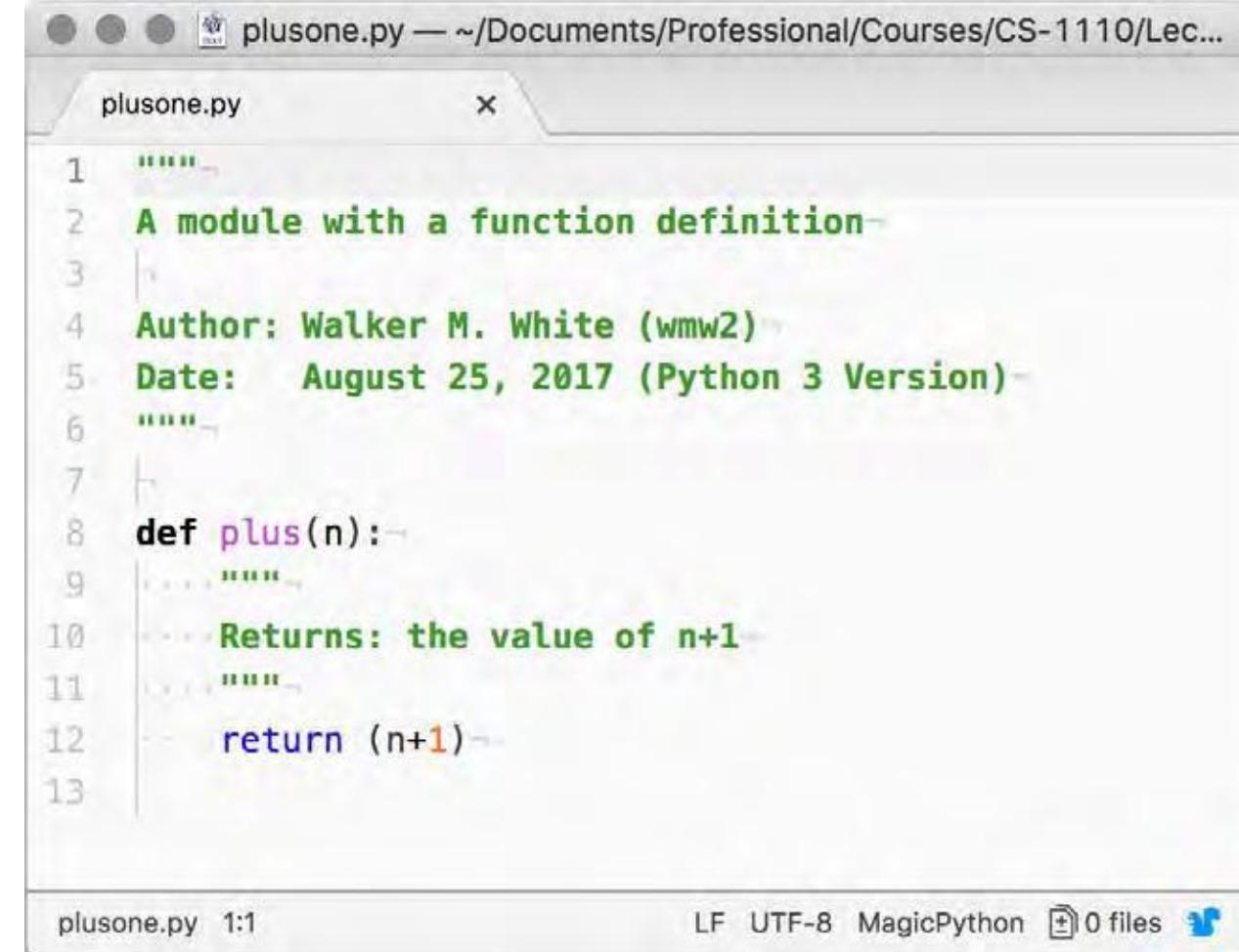
Next Time: Defining Functions

Function Call Function Definition

- Command to **do** the function
- Can put it anywhere
 - In the Python shell
 - Inside another module

```
modules — eCornell — python — 52x20
[>>> import plusone
[>>> plusone.plus(1)
2
[>>> plusone.plus(2)
3
[>>> plusone.plus(3)
4
>>> ]
```

- Command to **do** the function
- Belongs inside a module



```
plusone.py — ~/Documents/Professional/Courses/CS-1110/Lec...
plusone.py
1  """
2  A module with a function definition
3
4  Author: Walker M. White (wmw2)
5  Date: August 25, 2017 (Python 3 Version)
6  """
7
8  def plus(n):
9      """
10     Returns: the value of n+1
11     """
12     return (n+1)
```

plusone.py 1:1 LF UTF-8 MagicPython 0 files

Next Time: Defining Functions

Function Call Function Definition

- Command to **do** the function
- Can put it anywhere
 - In the Python shell
 - Inside another module

```
modules — eCornell — python — 52x20
[>>> import plusone
[>>> plusone.plus(1)
2
[>>> plusone.plus(2)
3
[>>> plusone.plus(3)
4
>>> ]
```

arguments
inside ()

Can **call** as many times as you want

- Command to **do** the function
- Belongs inside a module

```
plusone.py — ~/Documents/Professional/Courses/CS-1110/Lec...
plusone.py
1
2 A mod
3
4 Auth
5 Date
6
7
8 def plus(n):
9     """
10     Returns: the value of n+1
11     """
12     return (n+1)
13
```

But only define function **ONCE**



Thank you

